CS3003 Software Engineering Coursework - An Analysis of Software Development Metrics in Open-Source Projects

William Fitch - 1633241
Brunel University

Word Count: 1962

Section 1: Metric Selection & Justification (373/350 words)

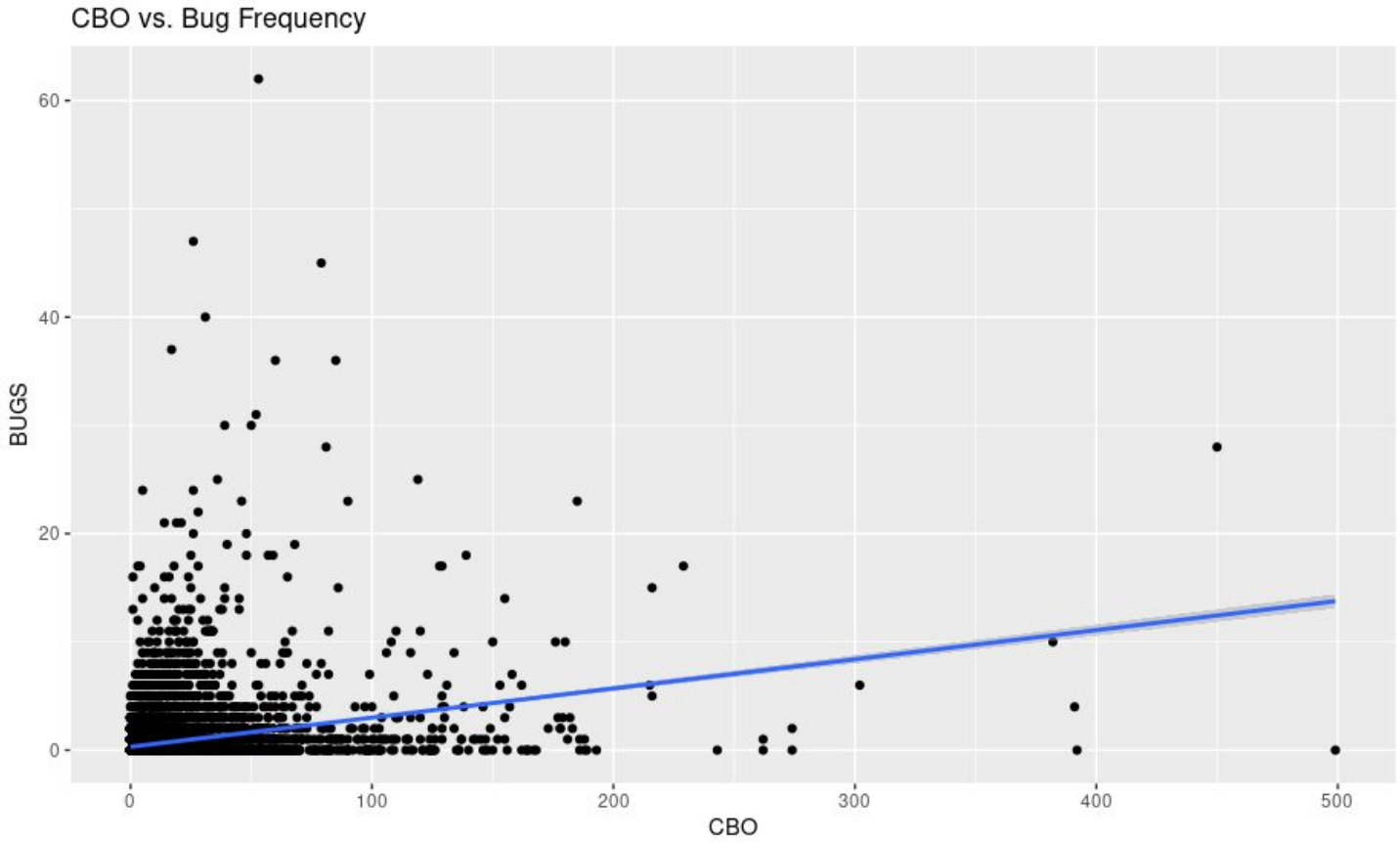
The attributes of good software are defined in Sommerville's Software Engineering 10th edition as "delivering the required functionality and performance to the user, and should be maintainable, dependable, and usable" (Sommerville, 2015). This assignment analyses a set of metrics from the given dataset, and evaluates their usefulness with respect to this definition of good software. By analysing these metrics, I intend to examine the effectiveness of software development best practices in reducing bugs per class, as well as their implementation from the projects' contributing guidelines. The metrics selected are primarily indicators of code quality and readability, describing code convolution, length, and obfuscation. As metrics that can verify software functionality, performance, or usability are unavailable to this investigation, there is no method for a comprehensive analysis of software quality to be undertaken from this dataset alone.

The metrics selected have some of the highest correlations to bug frequency in the dataset, as well as describing interesting possible relationships between themselves and bug density, from which conclusions about the relationship between code quality and software quality can be inferred. Based on preliminary investigations, the number of classes a class imports (fan.out) had the strongest correlation to the frequency of bugs in a class, implying that independent code is far more robust. Fan-out was also strongly correlated to the other metrics used. Based on its existing relationship to code understandability and bug density, Maximum Cyclomatic Complexity (MAXCC) was selected over average cyclomatic complexity as it is more likely to show the cyclomatic complexity of the sections of code that contain bugs, thereby being more representative of one cause of poor software. Coupling between objects (CBO) measures the number of links between objects, providing some measure of object-level readability and code quality through its adherence to common design practices such as SOLID principles (Martin, 2002). Though its effectiveness as a quality metric is dubious, LOC has been included to provide a standardised measure of quality at the line level, creating a metric of defect density and allowing direct comparisons of software quality between projects. All metrics in this analysis are compared against bug frequency per class, as this provides the most useful metric for the features described in the definition of good software in Section 1.

Section 2: Analysis and Discussion of Results (1589/1650 words)

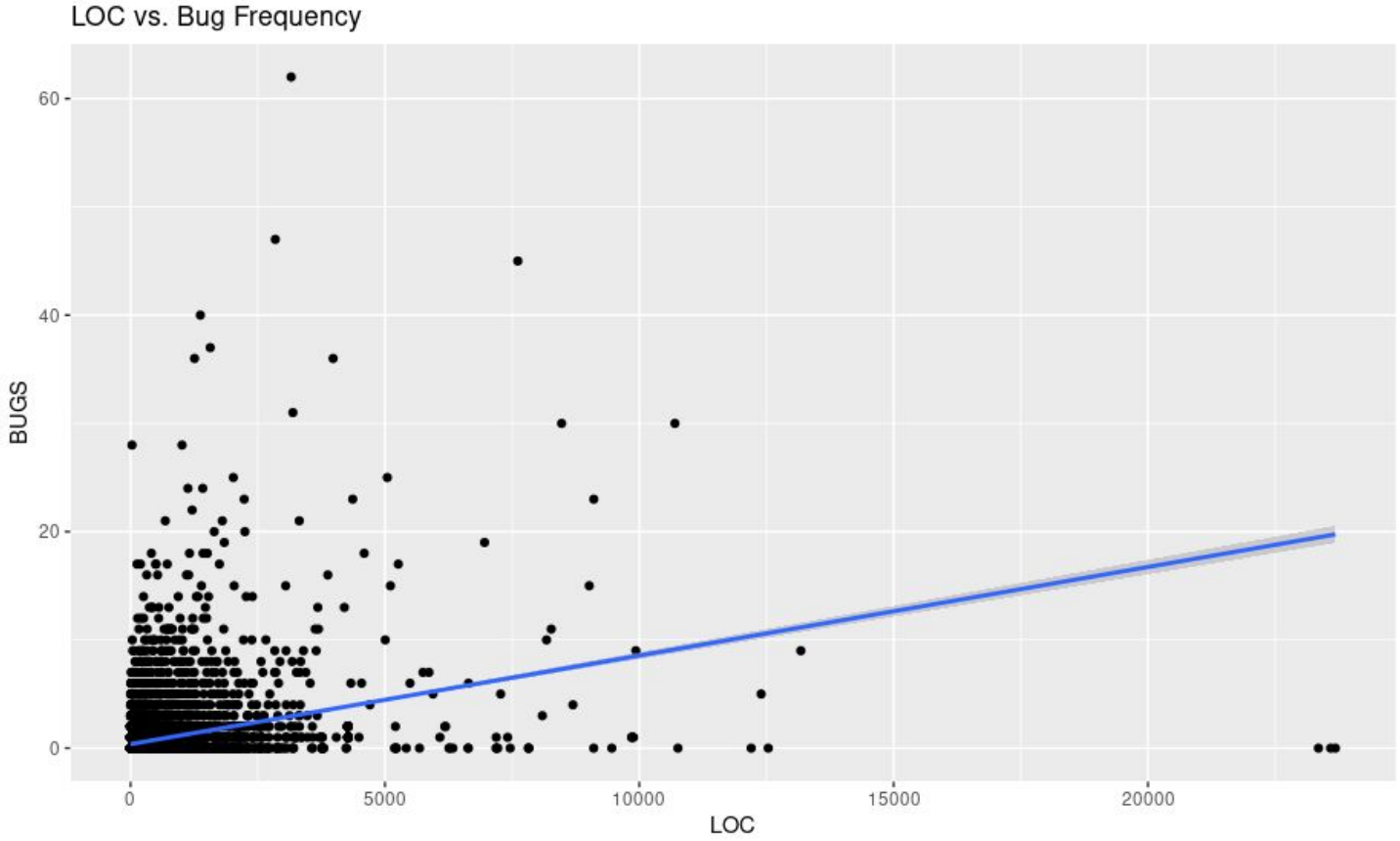
Individual Metric Analysis

Fig. 1 - Coupling Between Objects vs. Bug Frequency



As seen in Figures 1 & 5, CBO is weakly positively correlated with bug frequency. Coupling between objects reduces variable ownership, allowing free modification of object variables, which can in turn create bugs through race conditions or by editing variables that are currently in use by other active processes. It also obfuscates the code, as the relationships between classes and their objects are more poorly defined, reducing maintainability. This violates SOLID software development principles, and as shown above, is correlated with a higher rate of bugs per class. A loosely coupled object will encapsulate as much of its necessary functionality as possible, limiting its exposed interface and having better defined ownership over its variables. Loosely coupled object code follows SOLID software development principles, resulting in readable, well-structured code that can be easily maintained; in particular, it satisfies the single-responsibility, Liskov substitution, and interface segregation principles by ensuring that each class is responsible for its own methods and variables, and not allowing outside influence to dictate their use.

Fig. 2 - Lines of Code vs. Bug Frequency



In certain cases, when I investigated classes in this dataset that had no LOC but still had bugs, I found that these classes did in fact contain code, often being interfaces, or classes detailing data structures. Despite this vague definition of “lines of code”, and though it isn’t a highly effective metric in its own right, taking into account the lines of code in a class allows the calculation of the amount of bugs per line of code, creating a metric of code quality comparable across projects, used in Fig. 7. A larger codebase also requires more space and bandwidth to store and transfer the software, as well as more testing overhead costs.

As can be seen in Fig. 2, bugs are positively correlated to class size. This may be interpreted as larger classes having more bugs simply by virtue of their size; in actuality, Fig. 3 compares the lines of code in each class to the bugs per line of code, and shows how larger classes are more likely to be more stable than smaller classes, relative to the amount of code they contain. This is likely due to classes having more functionality as they increase in size, becoming a higher priority and having their bugs fixed faster. Further analysis using Feature Points may further illuminate this issue.

Fig.3 - Lines of Code vs Bugs per Line of Code, with outliers and bug-free classes removed.

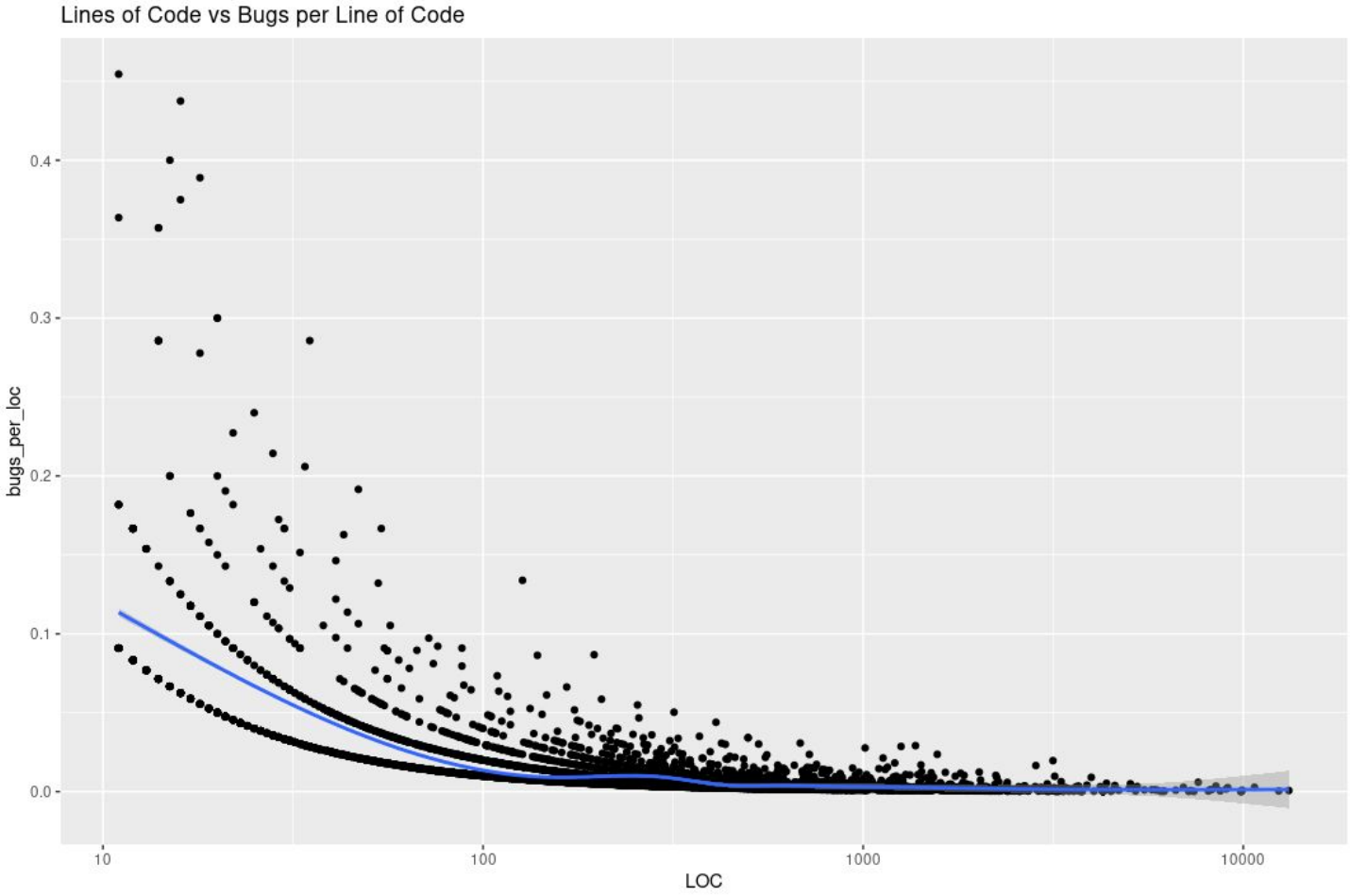
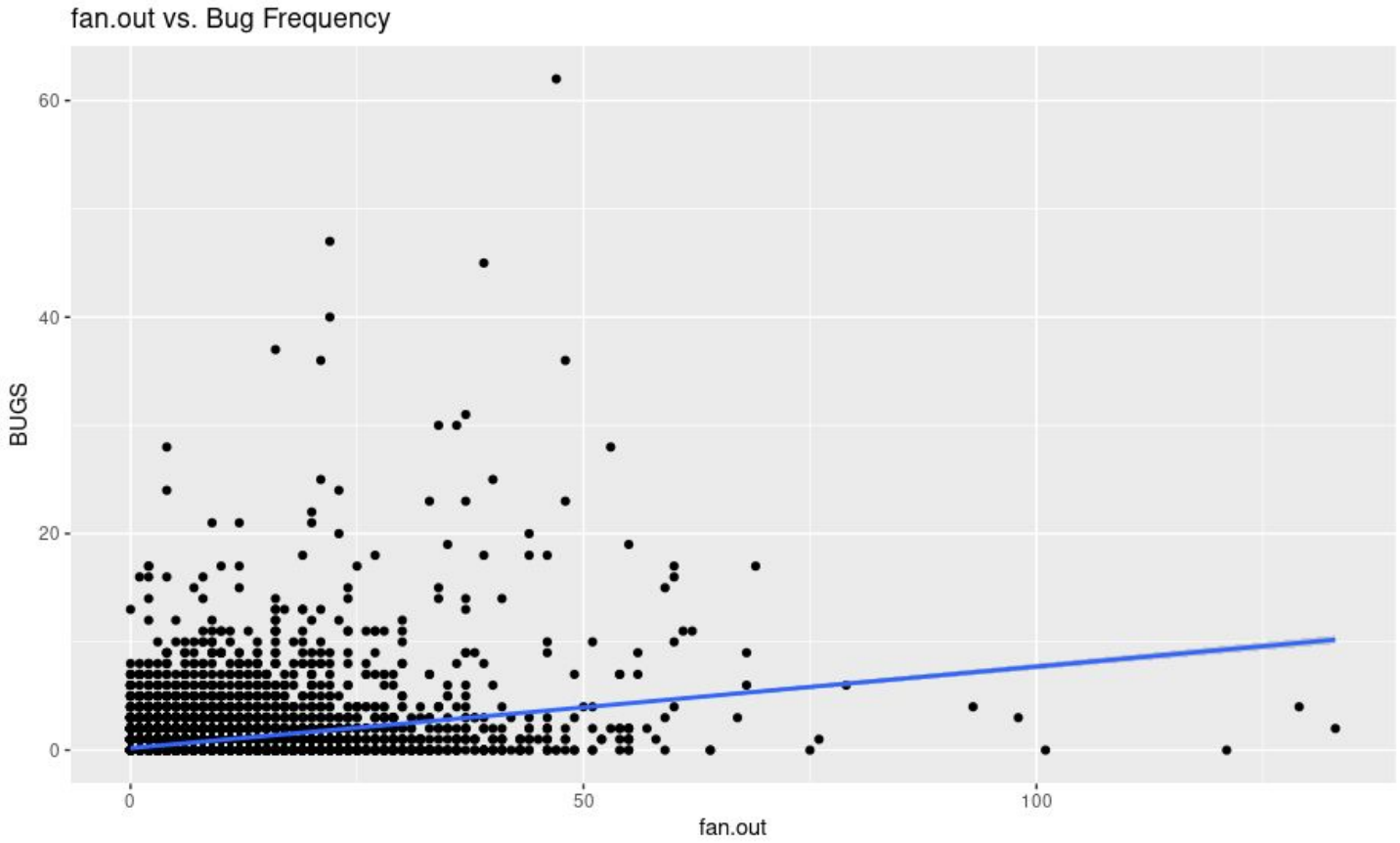


Fig. 4 - Fan-out vs. Bug Frequency



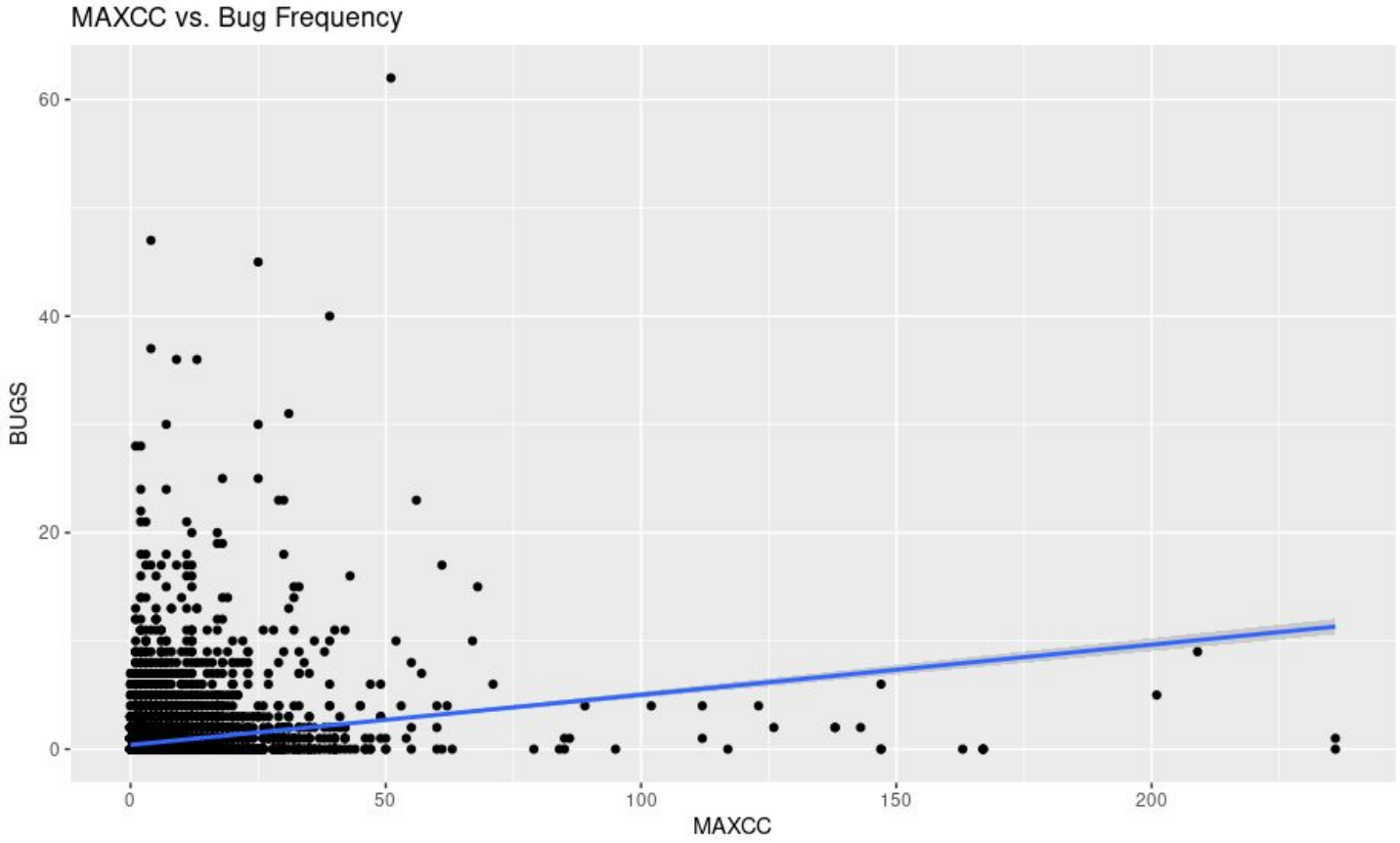
Similar to previous metrics, fan-out indirectly betrays code quality; as shown in Figure 4, code that imports large volumes of other modules or libraries is likely to be less coherent and understandable, as a large volume of imported code in a class requires a developer to understand the function of all these libraries if they are to be used effectively. If the imported code is unused, for example in a wildcard import in Java, this implies other poor coding practices. If the imported code is poorly understood, it may be used ineffectively, requiring more lines than necessary and prospectively causing further bugs. As development deadlines encroach, a developer may choose to take this shortcut temporarily, adding a characteristic “TODO: FIXME next week” comment to the code; these comments can still be found in legacy projects a decade or more later.

More imports may also imply a larger class, which may simply have more bugs by virtue of its size, as shown in Fig. 2, or a class that has more diverse functionality - for example, a Java class using the Java Swing framework may require more imports than a class that implements a data structure for a unique use case in the software. In this scenario, fan-out can be completely unrelated to bug density.

Importing code includes importing upstream bugs, possibly from unknown developers. External code may be poorly written, or may become unsupported at any time, creating a support vacuum that

must be filled if the library is to remain usable. If the library becomes deprecated, it must be replaced before it creates new bugs or security flaws. Utilising open-source dependencies allows a developer to audit the code before use, and maintain or extend it where necessary, such as further testing a specific edge case.

Fig. 5 - Maximum Cognitive Complexity vs. Bug Frequency



Code high in cognitive complexity doesn't directly create bugs, which is why this metric has the weakest correlation to bugs out of all of the chosen metrics; convoluted code is more difficult to read, making maintenance more difficult. As shown in Fig. 6 its correlation to other metrics that imply convoluted code further implies that a major source of bugs is poorly designed and maintained software; if code is arbitrarily complex, it is more difficult, more time-consuming, and less pleasant to work with. Given that this dataset is comprised of open-source projects primarily written by volunteers, confusing or unpleasant parts of a codebase may become neglected over time as developers choose to work on more enjoyable material. This may create a negative feedback loop as the project becomes outdated and develops more bugs, while developers leave the project for more exciting projects with fresh, well-written code.

Metric Interrelationship Analysis

Fig. 6 - Correlation coefficients between the measured metrics

Metric	CBO	LOC	fan.out	MAXCC	Bugs
CBO		0.2801242	0.5580192	0.2149597	0.2601886
LOC	0.2801242		0.3983838	0.4828345	0.3060179
fan.out	0.5580192	0.3983838		0.3476640	0.2983802
MAXCC	0.2149597	0.4828345	0.3476640		0.1949266
Bugs	0.2601886	0.3060179	0.2983802	0.1949266	

Looking at this matrix, two particularly strong correlations become apparent, between LOC and MAXCC, and between fan-out and CBO. A prospective cause for the correlation between code size and cognitive complexity is easy to suggest; longer code has a larger control flow graph with more space for complexity, and a developer that strives to minimise complexity in their code will also look to minimise length. The relationship between fan-out and coupling between objects may expose similar developer practices; a developer who imports large volumes of supplemental code may do so regardless of source, based on a development philosophy that imports whatever is necessary with little regard for program size or speed. This focus on implementation speed over finesse can also be seen in the correlations between fan.out, lines of code, and cognitive complexity - in fig. 7, we can see the project with the highest average fan.out value (Synapse) also has the highest MAXCC and CBO ratings.

Project-Based Analysis

Fig.7 - Project metadata ranked by bug density

Project	Total LOC	Total Bugs	Avg bugs/ KLOC	Total classes	Avg class length	Avg bugs/ class	Avg MAXCC	Avg fan-out	Avg CBO
jEdit	943208	943	0.9997795	3669	257.07495	0.2570183	4.916326	4.789316	9.560371
Ant	555417	637	1.1468860	2418	229.70099	0.2634409	3.887924	4.640612	9.190653
Ivy	199016	307	1.5425895	919	216.55713	0.3340588	3.060936	4.326442	9.454842
Xalan	1419715	2521	1.7757085	4142	342.76074	0.6086432	4.074119	6.410188	11.548527
Synapse	127260	265	2.0823511	656	193.99390	0.4039634	7.106707	8.657012	12.778963
Poi	411162	1377	3.3490449	1674	245.61649	0.8225806	3.175627	4.198327	8.372760
Xerces	594823	2044	3.4363163	1890	314.72116	1.0814815	3.423810	2.916931	5.811111
Camel	342363	1369	3.9986798	3511	97.51154	0.3899174	2.059242	5.697237	9.977784
Velocity	166139	729	4.3878921	707	234.99151	1.0311174	3.598303	5.704385	10.500707
Lucene	246697	1314	5.3263720	1177	209.59813	1.1163976	3.616822	4.481733	8.914189
Log4j	83099	645	7.7618263	555	149.72793	1.1621622	3.075676	3.387387	6.776577

All projects are written in Java and maintained by the Apache Software Foundation (ASF), with the exception of jEdit, which is independent. Comparing the code contribution method used by the ASF with those from the jEdit project, we can see jEdit uses a manual patch-based contribution method with more relaxed contributing standards, while Apache Ant uses a combination of patches and GitHub pull requests, along with a detailed style guide and an extensive CI backend. As seen in Figure 7, while Ant has slightly fewer bugs per KLOC, based on the predictions from the previous section its code is likely to be far cleaner. These results imply that allowing simple patch-based contributions to a project reduces the time taken to contribute to an open-source project, thereby improving stability by compromising on code quality. Apache Ant focuses on improving code quality through standardisation, PR reviewing and CI analyses, focusing on maintainability over stability.

When separated by project, certain macro trends become apparent. The perceived relationships between certain variables, such as MAXCC and bugs, at this scale appear to counter trends shown in previous examples on the scale of the whole dataset. This is likely due to an instance of the fallacy of averages or due to a small sample size of 11 projects.

Notes on Analysis, and Further Investigations

The given dataset skews heavily towards low bug frequency, as software is designed to be bug-free. This creates an external pressure for developers to fix bugs, especially in the most important classes, which may by their nature contain complex or legacy code that is unpleasant, but has been tested & fixed through 20 years of use. This long-term testing fosters a reluctance to refactor code that doesn't need changing - while it may not be perfectly written according to modern standards, it is reliable and extensively tested. This dataset is also deformed because it is based on open-source projects, meaning the majority of development on these projects is undertaken by volunteers in their spare time. Consequently, professional standards may be lower, corporate infrastructure support may not exist, and developers may neglect unpleasant or unnecessary areas of the codebase in favour of more exciting projects. These external factors (as well as natural variance) affect this analysis by weakening correlations and distorting regressions.

In conducting this analysis, certain conjectures have been made that would not be necessary with a broader dataset. For example, while bugs may be an indicator of stability, as one of the core components of the given definition for quality software, metrics directly measuring software stability would provide a more detailed picture of high-quality software; an example of this software stability could be the volume and diversity of crash reports, or bug failure rates found in CI tools. Similarly, information on bug severity and the time required to fix bugs may provide more insight into software maintainability.

Correcting methodological errors in this investigation would require going beyond correlational analysis towards more sophisticated statistical techniques that can correct for natural variance. Similarly, while some research was made on development practices in these areas, further research could analyse how these development practices have evolved over time. A historical analysis on the fixing of defective modules could further elucidate negative development practices using git history and legacy bug reporting logs.

Section 3: Bibliography

- Martin, R., 2002. *Agile Software Development: Principles, Patterns, And Practices*. Prentice Hall.
- Sommerville, I., 2015. *Software Engineering*. 10th ed. Pearson.